nesC

- Programming language for TinyOS and applications
- Support TinyOS components
- Whole-program analysis at compile time
  - Improve robustness: detect race conditions
  - Optimization: function inlining
- Static language
  - No function pointer
  - No malloc
  - Call graph and variable access are known at compile time
Application

- **Interfaces**
  - provides interface
  - uses interface

- **Implementation**
  - module: C behavior
  - configuration: select & wire

```c
module TimerP {
    provides {
        interface StdControl;
        interface Timer;
    }
    uses interface Clock;
    ...
}
```
interface Clock {
    command error_t setRate(char interval, char scale);
    event error_t fire();
}

interface Send {
    command error_t send(message_t *msg, uint16_t length);
    event error_t sendDone(message_t *msg, error_t success);
}

interface ADC {
    command error_t getData();
    event error_t dataReady(uint16_t data);
}

**Bidirectional** interface supports split-phase operation
module SurgeP {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}
implementation {
    bool busy;
    norace uint16_t sensorReading;
    async event result_t Timer.fired() {
        bool localBusy;
        atomic {
            localBusy = busy;
            busy = TRUE;
        }
        if (!localBusy)
            call ADC.getData();
        return SUCCESS;
    }
    async event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }
} ...
configuration TimerC {
    provides {
        interface StdControl;
        interface Timer;
    }
}

implementation {
    components TimerP, HWClock;

    StdControl = TimerP.StdControl;
    Timer = TimerP.Timer;

    TimerP.Clock -> HWClock.Clock;
}
Example: Surge

[Diagram of Surge system with various components and connections]

Chenyang Lu 31
Concurrent race: concurrent interrupts/tasks update shared variables.

Asynchronous code (AC): reachable from at least one interrupt.
Synchronous code (SC): reachable from tasks only.

Any update of a shared variable from AC is a potential race condition!
A Race Condition

module SurgeP { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;
    async event result_t Timer.fired() {
        if (!busy) {
            busy = TRUE;
            call ADC.getData();
        }
        return SUCCESS;
    }
    task void sendData() { // send sensorReading
        adcPacket.data = sensorReading;
        call Send.send(&adcPacket, sizeof adcPacket.data);
        return SUCCESS;
    }
    async event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }
}
Atomic Sections

atomic {
  <Statement list>
}

- Disable interrupt when atomic code is being executed
- But cannot disable interrupt for long!
  - No loop
  - No command/event
  - Function calls OK, but callee must meet restrictions too
module SurgeP { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;

    async event result_t Timer.fired() {
        bool localBusy;
        atomic {
            localBusy = busy;
            busy = TRUE;
        }
        if (!localBusy)
            call ADC.getData();
        return SUCCESS;
    }
}
nesC Compiler

➢ **Race-free invariant**: any update of a shared variable
  - is from SC only, or
  - occurs within an **atomic** section.

➢ Compiler returns error if the invariant is violated.

➢ **Fix**
  - Make access to shared variables **atomic**.
  - Move access to shared variables to tasks.
Results

- Tested on full TinyOS code, plus applications
  - 186 modules (121 modules, 65 configurations)
  - 20-69 modules/app, 35 average
  - 17 tasks, 75 events on average (per application) - lots of concurrency!

- Found 156 races: 103 real
  - About 6 per 1000 lines of code!

- Fixed races:
  - Add atomic sections
  - Post tasks (move code to task context)
Optimization: Inlining

<table>
<thead>
<tr>
<th>App</th>
<th>Code size</th>
<th>Code reduction</th>
<th>Data size</th>
<th>CPU reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inlined</td>
<td>noninlined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surge</td>
<td>14794</td>
<td>16984</td>
<td>12%</td>
<td>1188</td>
</tr>
<tr>
<td>Maté</td>
<td>25040</td>
<td>27458</td>
<td>9%</td>
<td>1710</td>
</tr>
<tr>
<td>TinyDB</td>
<td>64910</td>
<td>71724</td>
<td>10%</td>
<td>2894</td>
</tr>
</tbody>
</table>

- Inlining improves performance and reduces code size.
- Why?
Overhead for Function Calls

- **Caller**: call a function
  - Push return address to stack
  - Push parameters to stack
  - Jump to function

- **Callee**: receive a call
  - Pop parameters from stack

- **Callee**: return
  - Pop return address from stack
  - Push return value to stack
  - Jump back to caller

- **Caller**: return
  - Pop return value

*Many overhead instructions for function calls!*
Principles Revisited

- **Support TinyOS components**
  - Interface, modules, configuration

- Whole-program analysis and optimization
  - Improve robustness: detect race conditions
  - Optimization: function inlining
  - More: memory footprint.

- Static language
  - No malloc, no function pointers
Critiques

- No dynamic memory allocation
  - Bound memory footprint
  - Allow offline footprint analysis
  - How to size buffer when data size varies dynamically?

- Restriction: no “long-running” code in
  - command/event handlers
  - atomic sections
More

- Multi-threaded vs. event-driven architectures
  - A “standard” OS is more likely to be adopted by industry
  - Jury is still out…

- Alternative: Contiki
  - Multi-threaded OS
  - Plain C

- Alternative: Native Java Virtual Machine
  - Java programming
  - Virtual machine provides protection
  - Example: Sun SPOT
Reading


  - Purchase the book online
  - Download the first half of the published version for free.

- http://www.tinyos.net/